How to Build FFmpeg WebAssembly on Ubuntu 24.04

TechRounder PDF Edition

Live article: https://www.techrounder.com/linux/how-to-build-ffmpeg-webassembly-on-ubuntu-24-04/

By Vipin PG | Published March 26, 2026 | Updated March 26, 2026 | Format: Guide | 8 min read

Quick answer

If you have ever wanted to run video processing directly inside a web browser - without uploading anything to a server - you are probably already thinking about FFmpeg and WebAssembly. Together, they make that possible.

If you have ever wanted to run video processing directly inside a web browser - without uploading anything to a server - you are probably already thinking about FFmpeg and WebAssembly. Together, they make that possible. This guide walks you through the entire process of building FFmpeg as a WebAssembly module on Ubuntu 24.04, from installing dependencies all the way to testing the output in your browser.

Whether you are building a browser-based video editor, an offline media tool, or just exploring what WebAssembly can do, this is the foundation you need.

What Is FFmpeg WebAssembly and Why Does It Matter?

FFmpeg is one of the most capable open-source multimedia frameworks ever built. It handles video transcoding, audio manipulation, format conversion, streaming, and much more. Traditionally, FFmpeg runs on servers or local machines - it is a native tool compiled for your operating system's architecture.

WebAssembly (WASM) changes that equation entirely. It is a binary instruction format that runs inside web browsers at near-native speed, and it was designed to be a compilation target for languages like C and C++. Since FFmpeg is written in C, it can be compiled into a '.wasm' module and executed directly in the browser.

The practical benefits are significant. You can process video and audio on the client side without sending files to a server, build offline-capable media applications, reduce infrastructure costs, and deliver richer interactivity. ffmpeg.wasm - the community project that brings FFmpeg to the browser - is proof of how far this approach has come.

System Requirements Before You Start

Before diving into commands, make sure your Ubuntu 24.04 machine is ready for what is ahead. The compilation process is resource-intensive.

- Ubuntu 24.04 LTS (fresh install or fully updated)
- At least 8 GB of RAM
- At least 10 GB of free disk space
- A stable internet connection for downloading sources
- Basic familiarity with the Linux terminal

If your system meets these requirements, you are ready to proceed.

Step 1 - Update Your System

Always start with a clean, updated system. This prevents version conflicts and ensures you are working with the latest packages from Ubuntu's repositories.

```
sudo apt update && sudo apt upgrade -y
```

Once the update completes, reboot if a kernel update was applied.

Step 2 - Install Build Dependencies

FFmpeg's compilation requires several development tools and libraries. Install all of them in one shot using the following command:

```
sudo apt install -y git cmake build-essential python3 python3-pip autoconf automake libtool pkg-config yasm nasm
```

Here is what each package contributes to the process:

- git - for cloning the FFmpeg and Emscripten source repositories
- cmake - a build system required by several dependencies
- build-essential - includes 'gcc' , 'g++' , and 'make'
- python3 / python3-pip - required by the Emscripten SDK
- autoconf / automake / libtool - needed for running './configure' scripts
- pkg-config - helps locate library metadata during configuration
- yasm / nasm - assembler tools used by FFmpeg's native build (disabled for WASM, but still useful for environment validation)

Step 3 - Install Emscripten (the WASM Compiler Toolchain)

This is the most critical part of the setup. Emscripten is the compiler toolchain that translates C/C++ code into WebAssembly. It provides its own compiler frontend called 'emcc', which works much like 'gcc' but targets the browser instead of your native architecture.

Clone the Emscripten SDK from GitHub and set it up:

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

Verify the installation worked by checking the compiler version:

```
emcc -v
```

You should see version information printed to the terminal. If you get an error, make sure you sourced 'emsdk_env.sh' correctly - that script adds 'emcc' and related tools to your shell's 'PATH'.

> Quote: Tip: To avoid re-running the 'source' command every time you open a terminal, add it to your '~/.bashrc' or '~/.profile' file.

Step 4 - Clone the FFmpeg Source Code

Navigate back to a working directory and clone the official FFmpeg repository:

```
cd ~
```

```
git clone https://git.ffmpeg.org/ffmpeg.git ffmpeg-wasm
cd ffmpeg-wasm
```

If you prefer a specific release tag instead of the latest commit, you can check out a stable version:

```
git checkout n6.1
```

Using a tagged release is generally a better idea for production builds since it guarantees reproducibility.

Step 5 - Configure FFmpeg for WebAssembly Using Emscripten

This is where things get interesting. You cannot use the standard ‘./configure‘ directly - instead, you wrap it with ‘emconfigure‘, which tells the build system to use Emscripten’s toolchain instead of native compilers.

First, set your compiler flags. These tell Emscripten how to handle memory, threading, and optimization:

```
CFLAGS="-s USE_PTHREADS=1 -O3"
LDFLAGS="$CFLAGS -s INITIAL_MEMORY=33554432"
```

The ‘INITIAL_MEMORY=33554432‘ value sets the initial memory allocation to 32 MB. You can increase this if you plan to process larger files. The ‘-O3‘ flag enables aggressive code optimization, which significantly reduces the final binary size.

Now run the configuration:

```
emconfigure ./configure \
--target-os=none \
--arch=x86_32 \
--enable-cross-compile \
--disable-x86asm \
--disable-inline-asm \
--disable-stripping \
--disable-programs \
--disable-doc \
--extra-cflags="$CFLAGS" \
--extra-cxxflags="$CFLAGS" \
--extra-ldflags="$LDFLAGS" \
--nm="llvm-nm -g" \
--ar=emar \
--as=llvm-as \
--ranlib=llvm-ranlib \
--cc=emcc \
--cxx=em++ \
--objcc=emcc \
--dep-cc=emcc
```

Let’s break down the important flags here:

- -target-os=none - prevents FFmpeg from applying any OS-specific configurations, which is necessary for a portable WASM build

- -arch=x86_32 - keeps architectural optimizations minimal and compatible

- -enable-cross-compile - signals that we are building for a different target environment than the host

- -disable-x86asm / -disable-inline-asm - x86 assembly is incompatible with WebAssembly; both must be explicitly disabled

- -disable-stripping - Emscripten cannot strip native binaries, so this prevents a build failure

- -disable-programs - we only need the libraries, not the ‘ffmpeg‘ , ‘ffprobe‘ , or ‘ffplay‘ command-line tools

- -cc=emcc / -cxx=em++ - directs the build to use Emscripten’s compiler instead of GCC

Step 6 - Compile FFmpeg

With the configuration complete, kick off the build:

```
emmake make -j$(nproc)
```

The ‘$(nproc)‘ flag automatically uses all available CPU cores to speed up compilation. Depending on your machine, this can take anywhere from 10 minutes to over an hour. Be patient - this is a large codebase being cross-compiled into a completely different target format.

If you encounter an error during compilation, the most common culprits are:

- Forgetting to source ‘emsdk_env.sh‘ before running ‘emmake‘

- Running a configuration without the ‘--disable-inline-asm‘ flag (inline x86 assembly breaks the WASM build)

- Missing system packages from Step 2

Step 7 - Generate the WebAssembly Output File

Once the build finishes, you need to link the compiled objects into an actual ‘.wasm‘ module along with a JavaScript wrapper. Create an output directory and run the linking step:

```
mkdir -p wasm/dist
emcc \
-I. \
-O3 \
-s USE_PTHREADS=1 \
-s PROXY_TO_PTHREAD=1 \
-s INITIAL_MEMORY=33554432 \
-s EXPORTED_FUNCTIONS="[_main, _proxy_main]" \
-s EXPORTED_RUNTIME_METHODS=cwrap \
-o wasm/dist/ffmpeg-core.js \
fftools/ffmpeg_opt.c fftools/ffmpeg_filter.c fftools/ffmpeg_hw.c fftools/cmdutils.c fftools/ffmpeg.c \
libavcodec/libavcodec.a libavformat/libavformat.a libavfilter/libavfilter.a libavdevice/libavdevice.a
libavutil/libavutil.a libswscale/libswscale.a libswresample/libswresample.a
```

This command produces two files in ‘wasm/dist/‘:

- ffmpeg-core.js - the JavaScript glue code that bootstraps the WASM module

- ffmpeg-core.wasm - the actual WebAssembly binary

These two files work together. The ‘.wasm‘ file cannot be loaded directly by a browser on its own - it needs the JS wrapper to handle memory management, module initialization, and the bridge between JavaScript and C.

Step 8 - Test the Build Locally

You can do a quick sanity check by running the WASM output through Node.js before loading it in a browser:

```
node wasm/dist/ffmpeg-core.js -version
```

If you see FFmpeg version information printed to the terminal, your build is working correctly. This confirms that the WebAssembly module boots up and executes as expected.

For browser testing, set up a simple local server (Python’s built-in one works fine):

```
cd wasm/dist
python3 -m http.server 8080
```

Open your browser and navigate to ‘http://localhost:8080‘. Note that some WebAssembly features - particularly those using shared memory and threads - require specific HTTP response headers (‘Cross-Origin-Opener-Policy: same-origin‘ and ‘Cross-Origin-Embedder-Policy: require-corp‘) to work correctly. A production web server will need to be configured to send these headers.

Optimizing Your Build Size

One of the biggest practical concerns with FFmpeg WASM is file size. A full-featured build can easily exceed 30 MB, which is not acceptable for most web applications. Here are a few ways to keep things lean:

- Enable only the codecs you need. Use ‘--enable-encoder=libx264‘ and ‘--enable-decoder=h264‘ style flags during configuration to include only what your app requires. Avoid enabling everything by default.
- Use the ‘-O3‘ flag. This optimization level can cut binary size roughly in half compared to an unoptimized build.
- Disable unused libraries. Flags like ‘--disable-avdevice‘ and ‘--disable-postproc‘ remove components you may not need.
- Enable ‘--enable-small‘. This flag trades a small amount of performance for a reduced binary size and is worth including for browser deployments.

Browser Compatibility

WebAssembly is well-supported across all modern browsers. Chrome, Firefox, Safari, and Edge all handle WASM modules without issues. However, features like ‘SharedArrayBuffer‘ - which FFmpeg WASM uses for multi-threaded processing - require the specific security headers mentioned earlier. Without those headers, threading falls back to single-threaded execution, which affects performance on larger files.

For a deep dive into how the JavaScript-to-WASM bridge works and how the Emscripten file system API manages in-browser file reading and writing, the technical write-up by Jerome Wu on compiling FFmpeg with Emscripten is worth reading in full.

Common Errors and How to Fix Them

A few errors come up repeatedly when people attempt this build for the first time:

"invalid output constraint '=a' in asm" - This means inline assembly made it through the build. Make sure '--disable-inline-asm' and '--disable-x86asm' are both present in your configure arguments.

"file format not recognized" during strip - The strip tool cannot process WASM binaries. Add '--disable-stripping' to your configure arguments.

"emcc: command not found" - You forgot to run 'source ./emsdk_env.sh' in the current shell session. Re-run it and try again.

Extremely slow build or out-of-memory errors - Lower the '-j' value in 'emmake make -j4' to reduce parallel jobs, or add more swap space to your system.

For more detailed troubleshooting around codec dependencies like x264, this debugging walkthrough on FFmpeg with Emscripten and x264 pkg-config issues covers a frustratingly common pitfall in thorough detail.

What to Do With Your Build

Once you have a working 'ffmpeg-core.wasm' and its JS wrapper, you have a few paths forward depending on your project needs:

- Integrate into a frontend application - Load the WASM module from JavaScript, write input files to Emscripten's virtual file system using 'FS.writeFile()' , run FFmpeg operations by calling the exported 'main()' function with argument arrays, and read back results with 'FS.readFile()' .
- Use as a drop-in with ffmpeg.wasm - If you built a custom core, you can pair it with the ffmpeg.wasm JavaScript wrapper layer which provides a cleaner, higher-level API for browser use.
- Deploy via a CDN - Host your '.wasm' file on a CDN with the appropriate CORS and security headers to enable efficient loading across your user base.

Final Thoughts

Building FFmpeg for WebAssembly on Ubuntu 24.04 is not a five-minute task, but it is absolutely within reach if you follow each step carefully. The compilation process is thorough by design - you are essentially teaching a server-side media powerhouse how to live inside a browser sandbox.

The real reward comes once you have a working build: client-side video processing with no server dependency, no upload bottlenecks, and the full depth of FFmpeg's codec and filter capabilities available directly in your web app. That is a genuinely useful piece of infrastructure to have in your toolkit.

Take your time with the configuration flags, pay attention to the Emscripten environment setup, and do not skip the local testing step before integrating into a larger application.

References

1. github.com - ffmpegwasm / ffmpeg.wasm - https://github.com/ffmpegwasm/ffmpeg.wasm
2. emscripten.org - docs / getting_started - https://emscripten.org/docs/getting_started/downloads.html
3. jeromewu.github.io - build-ffmpeg-webassembly-version-part-2-compile-with-emscripten - https://jeromewu.github.io/build-ffmpeg-webassembly-version-part-2-compile-with-emscripten/
4. learntocodetogether.com - wasm-ffmpeg-emscripten-x264-pkg-config - https://learntocodetogether.com/wasm-ffmpeg-emscripten-x264-pkg-config/
5. dev.to - alfg / ffmpeg-webassembly-2cbl - https://dev.to/alfg/ffmpeg-webassembly-2cbl